



TREINAMENTO NODE.JS


Swagger

- O **Swagger** é uma ferramenta de código aberto que ajuda a documentar APIs de forma fácil e eficaz.
- Ele permite que você defina, construa, documente e consuma APIs RESTful.
- Com o Swagger, você pode descrever a estrutura da sua API, incluindo **endpoints**, parâmetros, tipos de dados de entrada e saída, e até mesmo fornecer exemplos de solicitações e respostas.
- <https://swagger.io/>



Swagger



 **swagger**

Select a spec default

Spring Boot REST API 1.0.0

[Base URL: localhost:8089/]
<http://localhost:8089/v2/api-docs>

“Spring Boot REST”

[Apache License Version 2.0](#)

question-controller

 Question Controller

GET

/questions

Mostra lista de questões

POST

/questions

createQuestion

PUT

/questions/{questionId}

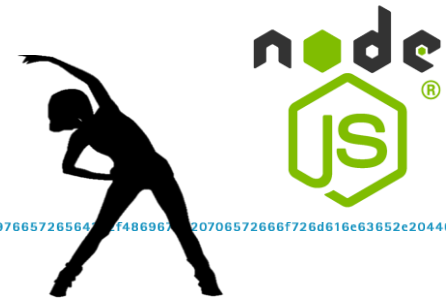
updateQuestion

DELETE

/questions/{questionId}

deleteQuestion

Atividade



- Adicionar o Swagger na API anterior do Sequelize – myApi
- Enviar evidências por email (tela funcionando)

Async/await

- A proposta das funções **async/await** é simplificar o uso de **promises**.
- A palavra **async** antes de uma função significa que a função sempre retorna uma promise.
- O operador **await** é utilizado para esperar por uma Promise. Ele pode ser usado **apenas** dentro de uma **async function**.
- Por exemplo, esta função async retorna uma promise resolved com a string “Ola Mundo”:

```
async function funcaoAsync() {  
    return 'Ola Mundo';  
}  
  
funcaoAsync().then(console.log);
```

```
async function funcaoAsync() {  
    return Promise.resolve('Ola Mundo')  
}  
  
funcaoAsync().then(console.log);
```



Async/await

- A sintaxe **async/await** reduz o amontoado de código em torno das promises.
- **Promises** foram introduzidas para resolver o famoso problema de *callback hell*, mas elas introduziram complexidade por si só, e complexidade sintática.
- Elas foram boas primitivas enquanto uma sintaxe melhor não surgia, então, quando chegou a hora certa, obtivemos as funções **async**.
- Elas fazem o código parecer síncrono, mas por baixo dos panos ele é assíncrono e não bloqueante.



Promises e Async/await

```
function funcSync() {
  console.log('Entrou em funcSync');
  return "Antes de tudo, olá mundo!";
}

async function funcAsync() {
  console.log('Entrou em "funcAsync".');
  return "Primeiro olá mundo!";
}

function outraFuncAsync() {
  console.log('Entrou em "outraFuncAsync".');
  return Promise.resolve("Segundo olá mundo!")
}

function novaFuncAsync() {
  console.log('Entrou em "novaFuncAsync".');
  return new Promise((resolve, reject) => {
    resolve("Terceiro olá mundo!")
  })
}

console.log('\nChamando funções para dentro das variáveis')
const texto0 = funcSync();           // "Antes de tudo, olá mundo!"
const texto1 = funcAsync();          // Promise { 'Primeiro olá mundo!' }
const texto2 = outraFuncAsync();     // Promise { 'Segundo olá mundo!' }
const texto3 = novaFuncAsync();      // Promise { 'Terceiro olá mundo!' }

console.log('\nImprimindo os valores das variáveis');
console.log( texto0 );
texto1.then( (valor) => { console.log(valor) } );
texto2.then( (valor) => { console.log(valor) } );
texto3.then( (valor) => { console.log(valor) } );
```

```
PS C:\curso-node> node .\index.js
```

Chamando funções para dentro das variáveis
Entrou em funcSync
Entrou em "funcAsync".
Entrou em "outraFuncAsync".
Entrou em "novaFuncAsync".

Imprimindo os valores das variáveis
Antes de tudo, olá mundo!
Primeiro olá mundo!
Segundo olá mundo!
Terceiro olá mundo!



Async/await

- Promise com **.then()** x **async/await**

```
var cep = '01001000';

const url = `https://viacep.com.br/ws/${cep}/json/`;

const response = fetch(url).then((data) => {
  data.json().then((endereco) => {
    console.log(endereco);
  })
});
```

```
var cep = '01001000';

const url = `https://viacep.com.br/ws/${cep}/json/`;

(async () => {
  const response = await fetch(url);
  const endereco = await response.json();
  console.log(endereco);
})();
```




Promisifying ("Promissificar")

- Essa técnica é um jeito de poder usar uma função **JavaScript** clássica que recebe uma **callback**, e retornar uma promise.
- Em versões recentes do Node.js, você não precisa fazer essa conversão manual para grande parte da API.
- Há uma função **promisifying** disponível no módulo **util** que fará isso por você, dado que a função que você esteja "**promissificando**" siga a assinatura correta.

Promisifying ("Promissificar")

- Transformando função **JavaScript** clássica em Promise

De forma manual

```
const fs = require('fs');

// Promisifying
const getFile = (fileName) => {
  return new Promise((resolve, reject) => {
    fs.readFile(fileName, (err, data) => {
      if (err) {
        reject(err)
        return
      }
      resolve(data)
    })
  })
}

// Using the function
getFile('./meuTexto.txt')
  .then(data => console.log(data.toString()))
  .catch(err => console.error(err));
```

Com a função promisify

```
const fs = require('fs');
const util = require('util');

// Promisifying fs.readFile
const readFileAsync = util.promisify(fs.readFile);

// Using the promisified function
readFileAsync('./meuTexto.txt')
  .then(data => console.log(data.toString()))
  .catch(err => console.error(err));
```



EventEmitter

- JavaScript no navegador tem uma grande quantidade de ações do usuário que são tratadas por eventos: clicks do mouse, teclas sendo pressionadas, movimentos do mouse, e muito mais.
- No lado do backend, o **Node.js** nos oferece a opção de trabalhar com um sistema similar usando o módulo **events**.
- Esse módulo, em particular, fornece a classe **EventEmitter**, que utilizamos para lidar com os eventos.
- Você a inicializa usando:

```
const events = require('events');  
const EventEmitter = new events.EventEmitter();
```



EventEmitter

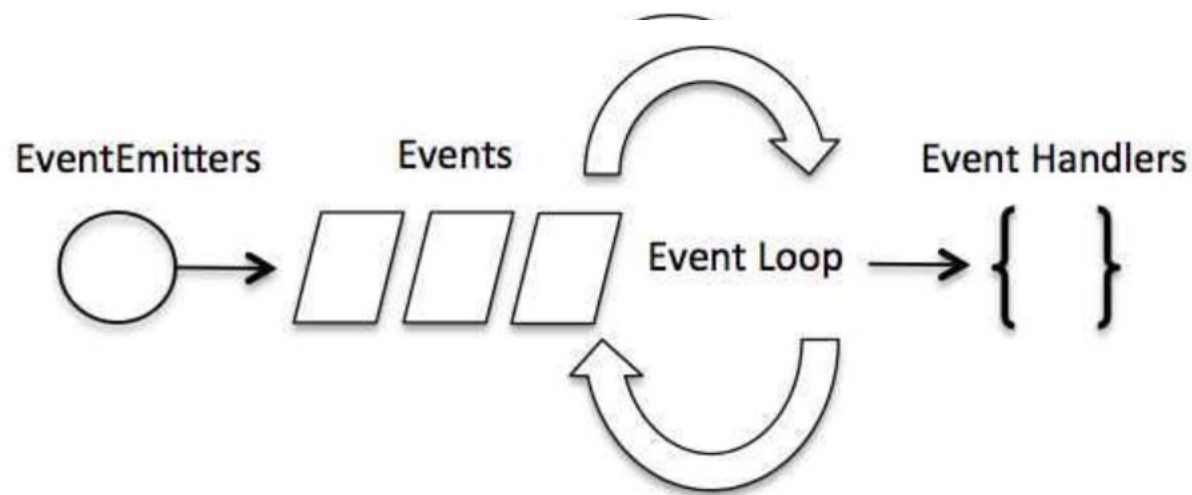
```
const events = require('events');  
const EventEmitter = new events.EventEmitter();
```

- Esse objeto **EventEmitter** expõe, entre muitos outros, os métodos **on** e **emit**.
 - **emit** é usado para acionar um evento
 - **on** é usado para adicionar uma função **callback** que será executada quando o evento for acionado

EventEmitter

Emissor de Eventos

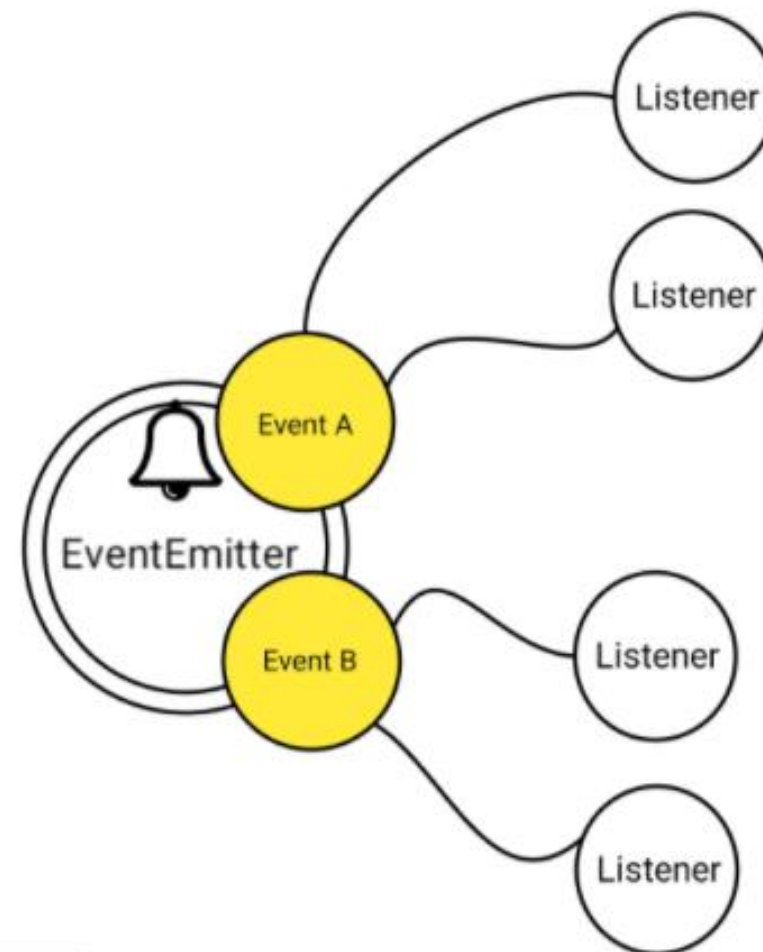
- Um Event Emitter é um objeto que emite notificações, ou eventos, em diferentes partes do seu código.
- Com um emissor de eventos, podemos simplesmente gerar um novo evento de uma parte diferente de um aplicativo e um ouvinte ouvirá o evento gerado e executará alguma ação para o evento;



EventEmitter

Eventos customizados

- **Event Emitters** são um dos conceitos mais importantes no Node, pois é com eles que emitimos eventos, que podem ser ouvidos por outros objetos no programa. Através dessa arquitetura é que podemos conectar pedaços de código;
- `const eventoEmissor = new EventEmitter();`





EventEmitter

// Cria o evento

```
var events = require('events');
```

```
const EventEmitter = new events.EventEmitter();
```

// Escuta o evento

```
EventEmitter.on('MeuEvento', () => {console.log('Informação recebida!!');} );
```

// dispara o evento

```
EventEmitter.emit('MeuEvento');
```

Node_evento1.js

Node_evento2.js